Production Scheduling in Almost Continuous Time*

Kevin R. Gue
George L. Nemhauser
School of Industrial & Systems Engineering
Georgia Institute of Technology

Mario Padron
AT&T Bell Laboratories

September 13, 1995

Abstract

We present a large scale production scheduling problem where each order is unique and the processing time for an operation can be close to the size of a time period. Because modeling the problem as a multiprocessor flowshop results in a computationally intractable formulation, we cast the problem in production planning terms and extract a production schedule from the solution. To solve our model, we introduce the notion of "almost continuous time" and show how to obtain good solutions to large problems efficiently.

1 Introduction

Production planning is a pivotal task for manufacturers in today's competitive marketplace. Effective production planning reduces labor and work-in-process inventory costs, lowers production costs by minimizing machine

*This research was supported by AT&T and NSF Grant DDM-9115768 to the Georgia Institute of Technology.

idle time, and increases the number of on-time job deliveries. Many of these planning problems are quite large, however, forcing firms to use techniques that offer little in the way of process optimization.

Our problem has characteristics of both production planning and flowshop scheduling, but standard approaches to both are unsatisfactory for it. Our remedy is to use appropriate elements of production planning and flowshop scheduling models to achieve good solutions.

Classical flowshop scheduling sequences a set of jobs on multiple machines such that each job is processed through the same order of machines. Machines may process only one job at a time and no job may be preempted. These problems are quite difficult to solve, except for the two machine case which is solved optimally using Johnson's rule. Problems with three or more machines are known to be NP-Hard [4].

Multiprocessor flowshop scheduling (MFS) is the more general problem that allows each processing stage to include a set of identical machines. Jobs must be scheduled on one machine in each stage. Typical objectives are to minimize the completion time of the last job or the number of late jobs. These problems are encountered in a variety of real world applications in manufacturing and other fields, but even small problems are difficult to solve optimally.

As in the area of flowshop scheduling, progress in algorithm development for MFS is constrained by its strongly NP-Hard complexity classification. All but the smallest problems are solved by heuristic methods. Unfortunately, many of the applications of this problem involve large numbers of jobs and machines. For example, our application has hundreds of jobs and dozens of machines.

Brah and Hunsucker [2] and Hariri and Potts [5] give branch-and-bound algorithms for MFS. They solve problems of up to 25 jobs and several machines. Heuristic methods for larger problems have also been proposed. Sriskandarajah and Sethi [10] give average and worst case results for several heuristics. Each of the heuristics reviewed addresses the problem of minimizing makespan. Wittrock [14] solves the multiprocessor flowshop problem with the objective of minimizing both makespan and buffer queue lengths.

Unlike combinatorial scheduling, production planning is mainly concerned with how much of a product to make in each time period to meet a set of demands. Planners must minimize work-in-process inventory and setup time delays while still meeting demand in each period. There is an extensive

literature on production planning problems; see Nam and Logendran [7] for a survey.

Billington, McClain and Thomas [1] give a mathematical programming formulation of the capacity constrained MRP system. Padron [9] solves a linear programming formulation of the capacity constrained multi-item, multi-stage production scheduling problem and compares the results with those he obtains with a more efficient decomposition procedure. His formulation is an alternative to MRP II methods for solving production scheduling problems and is similar in concept to the model we present. However, orders in MRP problems are typically for many units of a given product and the processing time of a unit is small with respect to a time period.

Among flowshop production problems, Chang and Liao [3] address the production scheduling of several part types through a multiprocessor flowshop. Using a Lagrangian relaxation and decomposition algorithm, they give a production plan for each part type to meet a set of demands. Their problem differs from others in that they do not schedule each machine, but each machine *group*. Luh and Hoitomt [6] use a Lagrangian relaxation decomposition scheme to solve large scale production scheduling problems with capacity constraints. Their technique, which employs a heuristic to meet capacity constraints, appears to yield good solutions while also providing a lower bound on the optimal objective function value. They observe that their method may not be suitable for problems with jobs requiring many operations on multiple machines.

Wein and Chevalier [12] discuss the relationships among sequencing, setting due-dates, and releasing jobs to the shop floor, and show how these decisions affect a job's due-date lead time (due-date minus arrival time) and tardiness. These issues are especially relevant to our application. The authors apply a two-step job release and priority sequencing policy to a two machine job shop, and compare their results with due-date policies.

The problem we address is a mixture of the MFS and production planning problems. It is like the MFS problem in that each production stage has several machines and can process many jobs at a time. Also, each job has an associated due date and set of processing times. Unlike the MFS problem, however, we do not seek an assignment of jobs to individual machines. Rather, we want to know only what jobs should be done in which stages for every time period to minimize the number of late jobs. Also like production planning models, our model seeks to minimize work-in-process

inventory costs.

We make a twofold contribution in this paper. First, we give a method for modeling and solving this hybrid production planning–flowshop scheduling problem. We show how to use our model to solve large real world problems in a reasonable amount of time. Second, we introduce to production scheduling a way to express "almost continuous time" in multi-period models through the use of special ordered sets type 2 (S2 sets), a technique used in integer programming to model piecewise linear functions. By almost continuous time we mean a compromise between discrete time where jobs are assigned to time periods and continuous time where a job can start at any instant.

We give the problem setting in Section 2. In Section 3 we present the model, give the formulation and discuss the use of S2 sets. Section 4 gives different solution methods and a set of example problems run on actual plant data. We conclude by discussing implementation issues.

2 Problem setting

The factory makes individual products by varying a number of product parameters to meet customer specifications. Since each order is unique, no products are made for inventory—all are made to order. Several hundred products are made each week, each having a due date that can vary from a few days to a few months in the future. Management must continually determine which orders to work in each production stage to provide the best service to its customers.

Each product goes through a series of production stages, where a stage consists of a number of identical parallel machines, as in Figure 1. Each stage performs only one operation on a job, although in some stages the nature of the product allows more than one machine in a stage to work the job simultaneously. A given job need not be worked in every stage, but all jobs follow the same relative sequence of machines. Some of the stages involve product testing, which results in rework for those products failing a test.

Production time in each stage varies by order, depending on parameters specified by the customer. Moreover, each order consumes a different amount of stage capacity, again depending on the parameters. Varying production times, stage capacities, the possibility of rework, and tight due dates produce a very difficult scheduling environment.

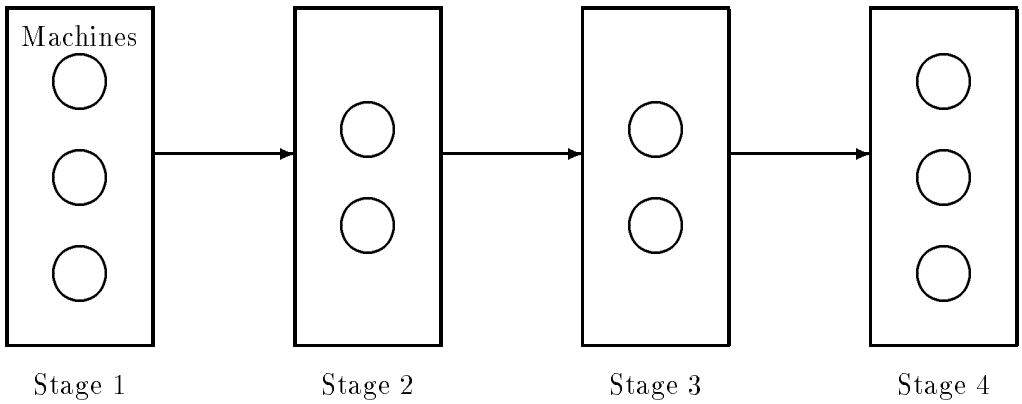

Figure 1: The production process

3 A production scheduling model

Within each stage there are scheduling issues that make formulating the entire problem very difficult. Even with an accurate formulation, the model's size would preclude solution. For these reasons we approach the problem from a slightly higher level, using aggregation in two areas.

First, we aggregate machines within a stage. Detailed sequencing and assignment issues within a stage, such as setup times, are left to supervisors or to other scheduling models. In our model, the capacity of a stage reflects the number and capabilities of the machines in that stage. The allocation of a job to a stage is only limited by stage capacity and the requirement that all earlier stages have been completed.

Next, we aggregate in time by creating time periods. Instead of specifying exact starting times for each job through its series of stages, we merely identify the time period in which to start the job in each stage. We make each period large enough with respect to job processing time that most jobs can be wholly done in each stage in one time period. However, we do not restrict the job to be completed in the time period in which it was started. So long as it is not interrupted, a job may be processed over two adjacent periods. We call this approach *almost continuous time.*

Definition *A schedule satisfies almost continuous time when any operation started in time period t finishes in period t or $t+1$.*

Job (i)	x_{i1}	x_{i2}
1	1.0	0
2	1.0	0
3	1.0	0
4	0.4	0.6
5	0.1	0.9

Table 1: Example solution

In other words, we allow some jobs to "spill over" into the next time period if there is insufficient capacity to complete them in the current one. This allows a stage to start a job toward the end of a period, even if it cannot complete the job in that time period.

We use the term "almost continuous time" because solutions, while not spelling out starting times for jobs, do provide a sense of order in time. Table 1 gives a small example that we interpret as follows. Let x_{it} be the fraction of job i done in period t. In period 1, jobs 1-3 should be started first (simultaneously if there are 3 machines). When the first of them is done, job 4 starts since it should be 40% complete by the end of the period. Job 5 starts on the next available machine. Jobs 4 and 5 finish in period 2.

We have chosen this approach because neither continuous time models nor discrete models are suitable for our application. Selecting exact starting times for each job in a stage would lead to a computationally intractable formulation. Traditional multi-period models are not suitable for different reasons, depending on the size of the time period. If time periods are small, such that a job takes several periods in each stage, the number of variables tends to be too large. If the time period is larger, such that jobs are wholly done within a period, stages will be made artificially idle whenever there is insufficient time to complete another job. Also, completed jobs will be made to wait until the next time period before proceeding to the next stage.

3.1 Aggregation Issues

There are some important consequences of aggregating with respect to machines and time. We present these in order to expose the limitations and proper application of our model.

Processing times and period length Our model is most useful when the processing times of jobs are close to the length of a time period. If the processing times are too small, jobs will be artificially made to wait until the next time period before moving to the next stage. If processing times are too large, our model of almost continuous time becomes invalid. This characteristic of the model means that it is most effective when processing times for different stages are more or less uniform.

Stage capacity To determine the capacity of each stage we compute the total processing time of all machines in a stage. For example, if a stage has 5 machines and a period is 8 hours, we give that stage a capacity of 40 *machine-hours*. If we process 5 jobs, each taking 8 hours of processing time, the assumption is valid. If we assign 8 jobs that take 5 hours each, the model would view this as a valid assignment also. However, since we cannot preempt a job, and for this example we can only do a job on one machine at a time, the assignment is not feasible. In particular, three jobs will require 2 hours of processing in the next time period. In our application, if scheduling does not work out "exactly," we rely on a run of the model in the next time period to correct the discrepancies.

Time and machine-time Because of the way we compute stage capacity, a job may take more time to complete than there is time in a period. This highlights the difference between real time and our measure of capacity, machine-time. In the above example, the model could assign a job that consumes 12 machine-hours on one machine, even though the time period is only 8 hours long, since stage capacity is 40 machine hours. In our application, however, a single time period (usually an 8 hour shift) is sufficient to complete nearly all of the jobs.

3.2 Formulation

We use the following notation:

$$
\begin{aligned}
I &= \text{set of all jobs} \\
I_j &= \text{set of jobs to be scheduled in stage } j \\
J_i &= \text{ordered set of stages required by job } i
\end{aligned}
$$

$j_i^\star$ = the last stage required for job i

T = number of time periods

x_{ijt} = fraction of job i started in stage j during period t

w_{ijt} = fraction of job i completed through stage j in period t and held as work-in-process in stage j until period $t+1$.

a_{ij} = amount of stage j used by job i (in machine-time)

b_j = capacity of stage j

d_i = due period of job i

u_{it} = fraction of job i overdue in period t

α_{it} = penalty constant for lateness of job i in period t

β_{ijt} = penalty constant for work-in-process of job i in stage j in period t.

S_{ij} = $\{w_{ij0}, x_{ij0}, x_{ij1}, \ldots, x_{ijT}\}$, an ordered set.

Our formulation, similar to production planning models, includes material balance constraints and machine capacities. In addition to minimizing the lateness of jobs, we seek to minimize work-in-process inventory costs. The former satisfies customers, while the latter reduces production costs and congestion on the shop floor. We also impose a lead time of one period on the production in a period, meaning the fraction x_{ijt} started in period t will not be available until period $t+1$.

The formulation is

$$\text{Minimize} \qquad \sum_{i \in I} \sum_{t=d_i+1}^{T} \alpha_{it} u_{it} + \sum_{i \in I} \sum_{j \in J_i} \sum_{t=1}^{d_i} \beta_{ijt} w_{ijt}$$

$$w_{ij,t-1} + x_{ij,t-1} - x_{i,j+1,t} - w_{ijt} = 0 \qquad \forall\, i, j \in \{J_i \setminus j_i^\star\},\ t = 1, \ldots, T-1 \qquad (1)$$

$$w_{ij_i^\star,t-1} + x_{ij_i^\star,t-1} - w_{ij_i^\star t} = 0 \qquad \forall\, i, t = 1, \ldots, d_i - 1 \qquad (2)$$

$$w_{ij_i^\star,d_i-1} + x_{ij_i^\star,d_i-1} + u_{id_i} = 1 \qquad \forall\, i \qquad (3)$$

$$-u_{i,t-1} + x_{ij_i^\star,t-1} + u_{it} = 0 \qquad \forall\, i, t = d_i + 1, \ldots, T \qquad (4)$$

$$\sum_{i \in I_j} a_{ij} x_{ijt} \leq b_j \qquad \forall\, j, t = 1, \ldots, T-1 \qquad (5)$$

$$w_{ij0} + \sum_{t=0}^{T-1} x_{ijt} = 1 \qquad \forall\, i,j \in J_i \tag{6}$$

$$0 \le w_{ijt}, x_{ijt} \le 1 \qquad \forall\, i,j \in J_i,\ t = 0,\ldots,T-1 \tag{7}$$

$$0 \le u_{it} \le 1 \qquad \forall\, i, t = d_i,\ldots,T \tag{8}$$

S_{ij} satisfies almost continuous time $\qquad \forall\, i,j \in J_i, \tag{9}$

where w_{ij0} and x_{ij0} are given, for all i and j.

The first term in the objective function penalizes the late completion of a job through the weighted sum of lateness variables. The variable u_{it} does not exactly count the number of periods job i is overdue, because doing this would require integer variables. Rather, it indicates the amount of job i left undone in each period after its due period. The second term penalizes work-in-process inventory for all periods through the due period. Note that the work-in-process could be in the form of a job finished before its due period, i.e. $w_{ij_i^*t} = 1$ for $t < d_i$. The second term, therefore, causes jobs to start as late as possible and, once started, to finish as quickly as possible. This is in keeping with the factory's inventory reduction goals.

Equation 1 is similar to a material balance constraint of standard production planning models, and is the key to understanding our model. For the purpose of explanation, define a new variable y_{ijt} to be the fraction of job i *available* in stage j in period t. The fraction y_{ijt} is composed of the fraction that was held as work-in-process through period $t-1$, plus the fraction that was started in period $t-1$ and became available in period t (lead time is one period). This gives the equation $y_{ijt} = w_{ij,t-1} + x_{ij,t-1}$. But saying that y_{ijt} is available means we must decide whether to start processing it in the next stage, to hold it through the period as work-in-process, or to divide it and do both (see figure 2). This leads to the equation $y_{ijt} = x_{i,j+1,t} + w_{ijt}$. Combining these two equations gives equation 1.

For the equation corresponding to $t = 1$, w_{ij0} refers to the fraction of job i already completed in stage j and ready to move on at the beginning of solution time—it is analogous to initial inventory. The term x_{ij0} refers to the fraction of job i actually being processed in stage j at the beginning of solution time.

Equation 2 is the same constraint for the last stage, in all time periods before the due period d_i. It prohibits the movement of a job to a nonexistent

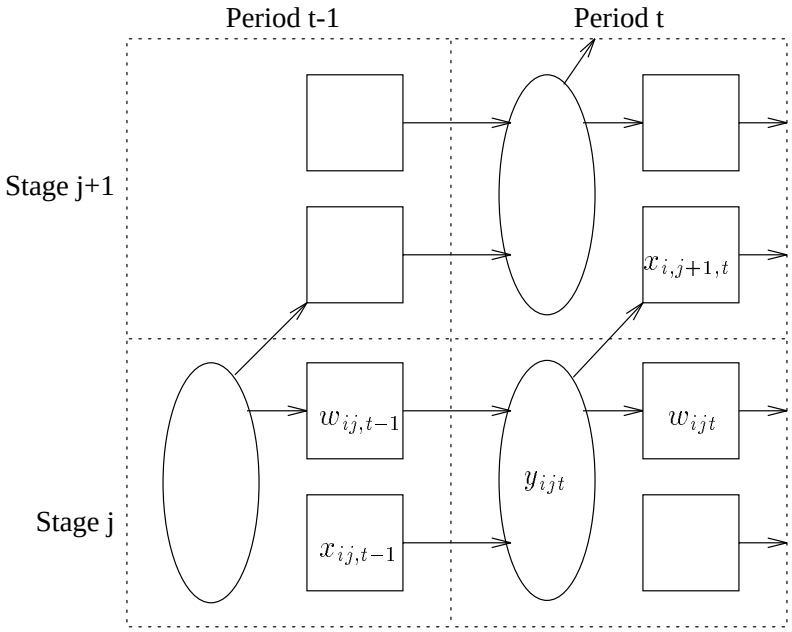

Figure 2: The model's material balance.

		Period		
Job	a_i	1	2	3
1	0.4	0.5		0.5
2	0.8	1.0		
3	1.0		1.0	

Table 2: x_{it} values violating almost continuous time

"next stage." Equation 3 introduces the lateness variable u_{it} and is written for the due period only. This constraint forces u_{it} to take a positive value if job i is overdue and left undone in period d_i. Equation 4 forces u_{it} to take a positive value for every time period after d_i for which the job is left undone. Constraint 5 is the capacity constraint for each stage in each time period, and equation 6 ensures that each job is done in the required stages. Condition 9 forces completion of a job in two periods once it starts in a stage.

To illustrate the importance of condition 9 we give a small example in Table 2. Consider the processing of 3 jobs on a machine in three time periods. Without condition 9, the solution is feasible, even though job 1 is split across periods 1 and 3. Job 3, which we assume was not available on this machine in period 1, preempts job 1 because it is more urgent. This leaves the remainder of job 1 to start in period 3. This solution is not acceptable for our application. A feasible solution is to start all of job 1 in period 3, leaving the machine idle for a small part of period 1.

3.3 S2 sets

An ordered set of decision variables meeting condition 9 is called a special ordered set type 2 (S2 set) and satisfies the following conditions.

- No more than two variables in the set are positive.
- If two variables in the set are positive, they are adjacent in the order.
- The sum of all variables in the set equals one.

S2 sets are commonly used to model nonlinear functions by piecewise linear approximation (see Tomlin [11] for an example), but they also provide a

	Problem								
	1	2	3	4	5	6	7	8	9
Jobs	42	51	64	86	127	134	255	333	402
Stages	7	7	7	7	7	7	7	7	7
Periods	12	12	12	12	15	15	17	25	25
S2 sets	145	177	223	294	441	374	886	841	1123
x_{ijt} variables	3071	3725	4700	6223	12425	11389	29201	48877	62975
Constraints	1655	1964	2435	3193	6315	6157	14638	25882	32808

Table 3: Description of test problems

natural way to enforce continuity of operations in production problems, where order is with respect to time.

Note that our model without condition 9 is a linear program and is readily solved using standard codes. Enforcing S2 sets can be done using branch-and-bound as we show in the next section.

4 Solution Methods

We provide three solution strategies for use in different contexts. First we give a branch and bound algorithm to solve the complete problem with all S2 sets defined. Next we relax the S2 sets to require almost continuous time only over the first period. This provides a way to solve larger problems while still retaining a clear implementation strategy. Finally we present solutions to the problem without defining S2 sets and show that these solutions are surprisingly easy to implement.

We solved a set of nine test problems using actual plant data. We altered the number of jobs and the machine capacities to demonstrate performance over a variety of problem sizes. Table 3 gives a summary of the problems. For all of the test problems we set $x_{ij0} = 0$, since $x_{ij0} > 0$ would mean $x_{ij1} = 1 - x_{ij0}$, and we could fix all the variables in that S2 set.

We solved our problems using MINTO, a Mixed INTeger Optimizer [8], running CPLEX 2.1 as the LP solver. We ran the algorithms on an IBM RISC 6000/550.

4.1 Branch and bound

An S2 violation exists if there is any fractional x_{ijt} that does not have an adjacent variable $x_{ij,t-1}$ or $x_{ij,t+1}$ equal to $(1 - x_{ijt})$. Suppose a set S_{ij} has an S2 violation and let x_{ijk} be the first violating fractional variable (i.e. the lowest k) in the set. We branch using the following equalities:

$$\textbf{Branch 1} \qquad \sum_{t=1}^{k} x_{ijt} = 0$$

$$\textbf{Branch 2} \qquad \sum_{t=k+2}^{T} x_{ijt} = 0$$

The S2 violation is cut off in both branches. Branch 1 cuts off the solution since x_{ijk} is positive, and branch 2 cuts it off since $x_{ijk} + x_{ij,k+1} < 1$ (otherwise S_{ij} would not have contained a violation).

We use this approach, but in addition use our knowledge of the problem environment to judiciously choose i and j. Before giving the branching scheme we make the following observation:

Observation *Fractional variables tend to occur in groups. That is, when a job is done in different periods in one stage, it usually does its remaining operations split over different periods as well.*

Close inspection of the constraint sets tells why this is so. Our constraints do not allow the model to do more than one "unit" of a job in any time period, regardless of which stage(s) does the processing. This restriction tends to "propagate" fractional variables in later stages when the variables take fractional values in an early stage (usually due to capacity shortage in a stage). Furthermore, the propagated fractions tend to take the same values, that is, a 60/40% split in periods 2 and 3 frequently precedes a 60/40% split in periods 3 and 4 in the next stage, etc. Our observation applies to S2 sets with violations as well as those without. Thus the propagated fractions could easily be 60/40% in periods 2 and 5, 60/40% in periods 3 and 6, and so on.

The observation also implies that correcting an early S2 violation will tend to correct other violations as well—the *correction* will be propagated. This leads to a branching rule that corrects early S2 violations (with respect to stages) before late ones. The key to our branching strategy lies in a proper

	Problem					
	4		5		6	
Rule	1	2	1	2	1	2
Nodes	147	∞	13	53	2669	∞
CPU sec	92	∞	44	75	3951	∞

Table 4: Comparison of two branching rules.

sort of the S2 sets. To take advantage of the observation we first sort the S2 sets by stage. After this is done we sort the sets again, this time by first stage (that stage that must process a job first) within each stage. This way, we examine S2 sets in earlier stages first, and among the jobs in a stage, we examine those that are there earlier in time before those that are there later. For example, the set $S_{i_1 j}$ comes before $S_{i_2,j+1}$ in the first sort. However, $S_{i_1 j}$ would come *after* $S_{i_2,j}$ if i_2's first stage is j and i_1's first stage is $j-1$.

Computational results indicate that branching order is important for large problems. Table 4 compares our branching strategy (Rule 1) with a "benign" one (Rule 2) on the largest problems solved to satisfy S2 conditions. For these runs, Rule 2 searches through S2 sets in an order specified only by the data input file. In this case we arranged jobs by number of stages left to process. For example, a job with only one stage remaining was considered before a job with three stages remaining. In this table, ∞ means the algorithm ran for over two hours without finding a feasible solution. The number of nodes indicates how many linear programs the algorithm solved during branch-and-bound. The results indicate the superiority of Rule 1.

4.2 Almost S2

In practice, we only need to know job assignments for the next time period, since the model will be run again after the period is over. This allows us to solve larger problems by ignoring S2 violations after, say, period 2 and working simply on "cleaning up" periods 1 and 2. Cleaning up the first two periods resolves any ambiguities in implementing the solution for the first period, and we ignore those in later periods.

Our branching scheme is the same as before except that we only correct violations that start in period 1. Thus if x_{ij1} is fractional and $x_{ij2} < 1 - x_{ij1}$,

	Problem								
	1	2	3	4	5	6	7	8	9
S2 sets	145	177	223	294	441	374	886	841	1123
CPU sec (LP)	1	1	3	4	12	14	83	373	814
Fractional Sets	30	32	60	67	59	90	151	155	246
Per. 1 violations	1	0	0	1	0	2	2	3	4
Total violations	6	0	3	13	15	15	46	65	75
CPU sec (Almost S2)	4	3	5	10	30	59	3943	1818	
Nodes in B-B tree	5	1	1	5	1	15	1437	119	
Fractional sets	33	32	55	65	64	88	145	159	
Total Violations	4	0	4	12	22	10	38	65	
CPU sec (Full S2)	7	3	15	92	44	3951			
Nodes in B-B tree	17	1	23	147	13	2669			
Fractional sets	30	32	54	74	78	109			

Table 5: Computational results for the test problems.

the branches are

$$\textbf{Branch 1} \qquad x_{ij1} = 0$$

$$\textbf{Branch 2} \qquad \sum_{t=3}^{T} x_{ijt} = 0.$$

4.3 LP solutions

At first glance it is not clear what solutions to the LP relaxation might look like. Highly fractional solutions, with fractional variables distributed throughout time are of little use. Surprisingly, solutions to the LP relaxation have quite a bit of structure and can be readily implemented. The first set of rows in Table 5 shows the number of S2 sets with fractions, the number with period 1 violations, and the total number of violations in the LP solutions to our test problems.

The high percentage of integer values in these solutions makes implementation possible. For these variables there is no ambiguity concerning the model's assignments. We can easily interpret those fractional variables in sets that satisfy the S2 set criteria. The only difficulty is interpreting those fractional variables that violate S2 requirements.

One implementation scheme is the following. For a given stage the solution to the LP relaxation gives a number of jobs that are to be completely worked in period 1 ($x_{ij1} = 1$). We do this set of jobs first, and among them, we give priority to those done in the next stage in period 2. Next we do those jobs with $x_{ij1} > 0$ and satisfying S2 conditions, followed by those not satisfying the S2 conditions.

Table 5 gives the computational results for all problems and solution techniques. A missing data entry indicates that CPU time would be more than one hour for that problem and solution technique. Note how few period 1 violations occur in even the largest problems. This characteristic of LP solutions allows us to solve large "almost S2" problems using branch-and-bound. Also, resolving period 1 violations does not necessarily reduce the number of fractional S2 sets, as seen in problems 3, 5, and 8, implying that resolving some violations creates others. Finally, we note that LP solutions have relatively few S2 violations—fewer than 8% of all S2 sets in each test problem contain violations.

It is interesting to ask *why* the LP solutions have such nice structure. Consider the recursion given by equation 1 for some $1 \leq t \leq T$. We have

$$\begin{aligned}
w_{ijt} &= w_{ij,t-1} + x_{ij,t-1} - x_{i,j+1,t} \\
&= (w_{ij,t-2} + x_{ij,t-2} - x_{i,j+1,t-1}) + x_{ij,t-1} - x_{i,j+1,t} \\
&= w_{ij,t-3} + \sum_{k=1}^{3} x_{ij,t-k} - \sum_{k=0}^{2} x_{i,j+1,t-k} \\
&\vdots \\
&= w_{ij0} + \sum_{k=1}^{t} x_{ij,t-k} - \sum_{k=0}^{t-1} x_{i,j+1,t-k} \\
&= w_{ij0} + \sum_{k=0}^{t-1} x_{ijk} - \sum_{k=1}^{t} x_{i,j+1,k}
\end{aligned}$$

But $w_{ijt} \geq 0$ gives the sequencing inequality

$$\sum_{k=1}^{t} x_{i,j+1,k} \leq w_{ij0} + \sum_{k=0}^{t-1} x_{ijk}$$

This inequality says that the amount of a job started in a stage through period t can be no more than that amount started in the previous stage

through period $t-1$. It is analogous to a family of cuts given by Wilson [13], but is not written explicitly in our model since the material balance equations already embody these cuts. These inequalities provide an ordering on the variables in time that is close to the S2 sets condition and explains the nice structure of LP solutions.

Formulating our problem in a different way, without these sequencing inequalities, confirms their importance. Solutions to this formulation are extremely fractional and difficult to interpret.

4.4 Implementation

We can run the model using any of the three solution strategies, depending on the user's requirements. If the problem is not too large, we run the full S2 branching to obtain the most unqualified solution. We can solve a larger problem using almost S2 branching and achieve a clear implementation for the current period. We solve larger problems as linear programs, and interpret the solution using a method such as that given in Section 4.3.

We use our model on a rolling horizon basis, scheduling all jobs to completion on each run. A run begins with a snapshot of jobs in progress to determine the w_{ij0}'s and schedules each job through its remaining stages. During a shift however, new orders arrive, jobs may fail inspections, and machines may breakdown, making frequent runs of the model necessary. This is especially true when new orders have demanding due dates. A fresh run of the model can push lower priority or ahead of schedule jobs aside to make room for the more urgent work.

Our model serves two purposes. First, it determines which jobs to work in which stages on the current shift. For each area the model provides a list of jobs to complete and a list of jobs to start in the current shift. These lists can be used directly by shop floor supervisors, or as input to more detailed sequencing models.

Second, management can use the model to evaluate shop floor decision making. When a job is not finished in accordance with the model solution, management can investigate the reasons behind the schedule alteration ("We couldn't process that many jobs in our stage in one shift") and suggest changes to the model (reduce the capacity of that stage) or change operational procedures to function according to the model. This enables management to fine tune the model and their operations over time.

Our model is limited by the continuous processing of a job in a stage in at most two periods. However, we can accommodate the uninterrupted processing of a job over several periods by generating new branching rules.

Finally, we observe that our model of almost continuous time can be applied to jobshop scheduling simply by specifying the appropriate machine sequence for each job.

References

[1] Billington, P.J., McClain, J.O. and Thomas, J.L., "Mathematical Programming Approaches to Capacity-Constrained MRP Systems: Review, Formulation and Problem Reduction," *Management Science* 29, 1126-1141 (1983).

[2] Brah, S.A., and Hunsucker, J.L., "Branch and bound algorithm for the flow shop with multiple processors," *European Journal of Operational Research* 51, 88-99 (1991).

[3] Chang, S.C., and Liao, D.Y., "Scheduling flexible flow shops with no setup effects," *Proceedings–IEEE International Conference on Robotics and Automation*, 1179-1184 (1992).

[4] Garey, M.R., and Johnson, D.S., and Sethi, R., "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research* 1, 117-129 (1976).

[5] Hariri, A.M.A., and Potts, C.N., "Branch and bound algorithm to minimize the number of late jobs in a permutation flow-shop," *European Journal of Operational Research* 38, 228-237 (1989).

[6] Luh, P.B., and Hoitomt, D.J., "Scheduling of Manufacturing Systems Using the Lagrangian Relaxation Technique," *IEEE Transactions on Automatic Control* 38, 1066-1079 (1993).

[7] Nam, S., and Logendran, R., "Aggregate production planning – a survey of models and methodologies," *European Journal of Operational Research* 61, 225-272 (1992).

[8] Nemhauser, G.L., Savelsbergh, M.W.P., and Sigismondi, G.C., "MINTO, a Mixed INTeger Optimizer," *Operations Research Letters* 15, 47-58 (1994).

[9] Padron, M., "A Linear Programming Solution to the Capacity-Constrained MRP Problem," *Proceedings–International Industrial Engineering Conference*, 495-499 (1987).

[10] Sriskandarajah, C., and Sethi, S.P., "Scheduling algorithms for flexible flowshops: Worst and average case performance," *European Journal of Operational Research* 43, 143-160 (1989).

[11] Tomlin, J.A., "Special ordered sets and an application to gas supply operations planning," *Mathematical Programming* 42, 69-84 (1988).

[12] Wein, L.M., and Chevalier, P.B., "A broader view of the job-shop scheduling problem," *Management Science* 38, 1018-1033 (1992).

[13] Wilson, J.M., "Generating cuts in integer programming with families of special ordered sets," *European Journal of Operational Research* 46, 101-108 (1990).

[14] Wittrock, R.J., "An adaptable scheduling algorithm for flexible flowlines," *Operations Research* 36, 445-453 (1988).